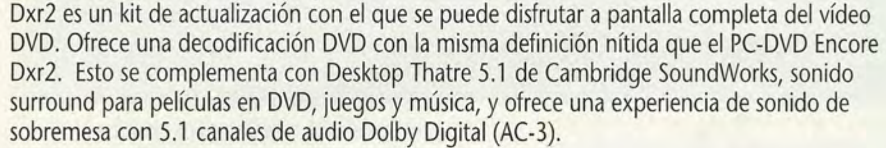


Sumario

- **3D Manía** 2
  Las pistas para usar correctamente los, cada vez más habituales, mundos en tres dimensiones.
- **DIV** 5
  Los secretos de este ejemplar entorno de desarrollo de videojuegos contados con amenidad para ayudaros en vuestras creaciones.
- **Curso Direct X** 9
  De la mano de nuestro equipo no tendréis ningún problema para manejar las famosas librerías de Microsoft
- **Taller 2D** 13
  Tercera entrega de este cursillo de animación muy práctico a la hora de ser utilizado en videojuegos.

PC-DVD Encore DXr2 y Desktop Theatre 5.1



Creative Labs y Cambridge SoundWorks han lanzado el primer sistema de altavoces desktot theatre para PC. El nuevo PC-DVD Encore 5X con tecnología DXr2 y los sistemas DeskTop Theatre 5.1 son una combinación idónea de la tecnología PC-DVD y el sonido surround Dolby Digital, gracias a la cual se consigue una experiencia recreativa en el Pc que hasta ahora sólo estaba disponible a un alto coste.
La tecnología de mejora de imágenes PC-DVD Encore 5X con Dxr2 es un kit de actualización con el que se puede disfrutar a pantalla completa del vídeo DVD. Ofrece una decodificación DVD con la misma definición nítida que el PC-DVD Encore Dxr2. Esto se complementa con Desktop Thatre 5.1 de Cambridge SoundWorks, sonido surround para películas en DVD, juegos y música, y ofrece una experiencia de sonido de sobremesa con 5.1 canales de audio Dolby Digital (AC-3).

Maxi Studio ISIS





Giullemot acaba de presentar una nueva tarjeta de sonido que a buen seguro entusiasmará a todos aquellos que gustan de confundir a su PC con un piano, una batería o similar. Con un precio sumamente atractivo, un poco menos de 60.000 pesetas, podemos obtener este interesante producto que, a partir de ahora, habremos de considerarlo como un paso más en la búsqueda de la sintetización musical. Las características de Maxi Studio Isis son las que a continuación enumeramos: 8 entradas y 4 salidas Full Duplex, sintetizador de gran calidad, entrada y salidas digitales y convertidores de 20-bit, compatibilidad con los principales programas musicales (como Cakewalk y Cubevase VST) y, por supuesto, la inclusión del Logic Audio Pro ISIS.
Está basada en la tecnología RISC (calidad MIDI ultrarrealista que no toma recursos de la CPU). Los sintetizadores son GM/GS de 64 voces y sampler estéreo multicapas de 48 KHz. Incluye 2 LFO (Low Frecuency Oscillators), 3 Egs (Envelope Generators) y dos mesas de tracking. Filtro de resonancia de hasta 24 dB. Memoria ampliable a 36 MB de Ram, las entradas habituales y 2 salidas estéreo para conectar cuatro altavoces.
También viene con un Rack externo con 8 entradas y 4 salidas analógicas de convertidores ADC y DAC de 20 bit, entrada y salidas digitales S/PDIF con conexiones RCA chapadas en oro y conexiones ópticas para transferir tus programas musicales con calidad CD a un DAT o a un mini disc, más tres conectores MIDI.

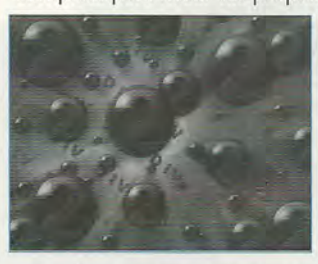


El software que acompaña a esta excelente opción incluye un numeroso grupo de programas destinados a completar esta tarjeta y hacer de tu ordenador un estudio completo. Entre otros, esta oferta incluye un paquete exclusivo diseñado por el departamento de investigación de Gullemot, demos de los mejores programas Audio/Midi del mercado, editor de audio, etcétera.

Por fin Div2, la herramienta



Con un código optimizado para Pentium, rutinas para manejo de textos, generador automático de sprites, mapeador de niveles para juegos 3D y un compilador más optimizado, llega Div2. Sigue la misma línea de calidad que su predecesor, pero la inclusión de todas estas novedades terminarán por convertirlo en una herramienta mucho más personal al tiempo que más accesible para los que aún no han saboreado el placer de programar sus propios juegos. Porque en eso consiste Div 2 Games Studio, una herramienta completa para hacer tus propios juegos con calidad comercial.



En esta ocasión, Div 2 enfoca parte de sus posibilidades en los juegos y entornos 3D, además mantiene una total compatibilidad con su antecesor e incluye toda una galería de sonidos, aplicaciones y juegos para ayudarte en tus propias creaciones.
Aprovechamos para recordaros que, en estas mismas páginas y en la revista Divmanía, podréis encontrar numerosas orientaciones y consejos para que llevéis a buen término, esto es, a juegos comerciales, vuestra inventiva.



Destacamos

Dentro de nuestro CD-Rom de portada incluimos en esta ocasión el siguiente material relacionado con la sección Game Developer:
- Los códigos fuente de los ejemplos comentados en el artículo 3D Manía.
- Los códigos fuente de los ejemplos comentados en el artículo DirectX.

Autor: Jesús de Santos
icg_ent@bigfoot.com

Introducción a los mapas de luces [LightMaps]

Dedicamos este mes una pequeña introducción a una técnica que goza de gran popularidad entre los juegos 3D actuales (sobre todo juegos de escenarios interiores).

Hasta la fecha, hemos revisado los siguientes métodos de iluminación (más bien deberíamos decir métodos de pintado, pues el efecto es el de pintar los elementos) de los polígonos:

1. *Flat*: el triángulo posee un color constante.
2. *Gouraud*: cada uno de los píxeles del triángulo posee el color correspondiente a la interpolación lineal (cuadrática funciona mejor a base de un mayor coste de cálculo) del color de los tres vértices.
3. *Phong*: se interpolan las normales de los tres vértices y con esa normal en cada vértice aplicamos el modelo de iluminación correspondiente.

No vamos describir a fondo estas técnicas pues se suponen conocidas por el lector. Si no es el caso, hemos dedicado algunos artículos para explicar estas técnicas. El motor 3D que hemos ido desarrollando paralelamente a estos artículos implementa sombreado *Gouraud* en los polígonos.

El sombreado *Gouraud* no puede pintar luces que caigan en el interior de los polígonos

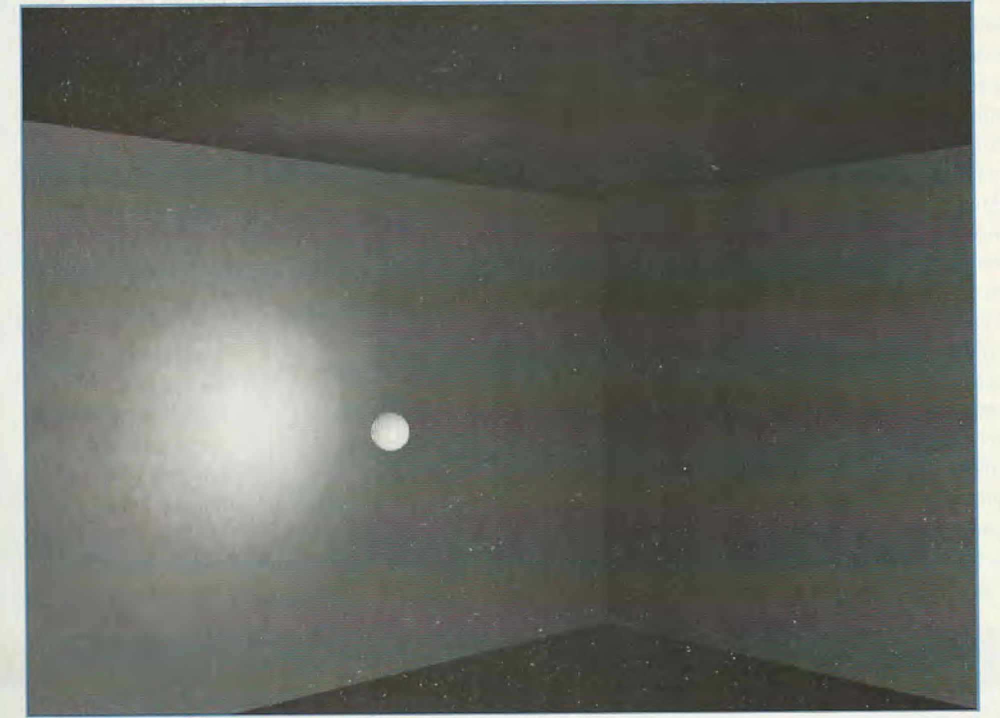
El problema que vamos a plantear este mes aparece claramente en las figuras 1 y 2. En ellas se muestra una habitación que tiene cuatro paredes y una luz interior de tipo *omni*. En la figura 1 tenemos el resultado que obtenemos si aplicamos un *render* de tipo *Phong*. El resultado es el correcto. En cambio, en la figura número 2 estamos aplicando un *render* de tipo *Gouraud*. El problema salta a la vista. No aparece el círculo de luz en el interior del polígono.

Este es un problema que ya señalamos en nuestro correspondiente artículo.

El sombreado *Gouraud* no es capaz de pintar luces que caigan en el interior de los polígonos. Básicamente, con *Gouraud* lo que hacemos es una interpolación lineal de la luz, pero en ningún momento estamos teniendo en cuenta la orientación que tenga el vector luz con respecto a la normal.

Así, por ejemplo, dado un vector normal y un vector luz, es indiferente si éstos forman un ángulo de 30 o bien de -30 grados (lo cual, desde un punto de vista puramente matemático, es perfectamente correcto, pues el producto escalar siempre devuelve el menor de los ángulos).

FIGURA 1. MUESTRA DEL RENDER DE UNA ESCENA CON INTERPOLACION PHONG. EL BRILLO ESPECULAR APARECE PERFECTAMENTE EN EL INTERIOR DEL POLIGONO.



Los mapas de luces se utilizan en muchos de los títulos actuales, como Quake

Teniendo en cuenta que esta escena (la que aparece dentro de las figuras 1 y 2) es muy corriente en muchos de los juegos que nos llegan en la actualidad, nos encontramos ante un problema muy serio. Así, por ejemplo, en arcades tridimensionales como los conocidos *Quake* o *Unreal* (y en todos los juegos actuales que se desarrollen en interiores) observamos cómo las luces se reflejan perfectamente en las paredes. Sin embargo, si nos preocupamos por editar algunos de los niveles de estos programas observaremos que la complejidad estructural es mínima.

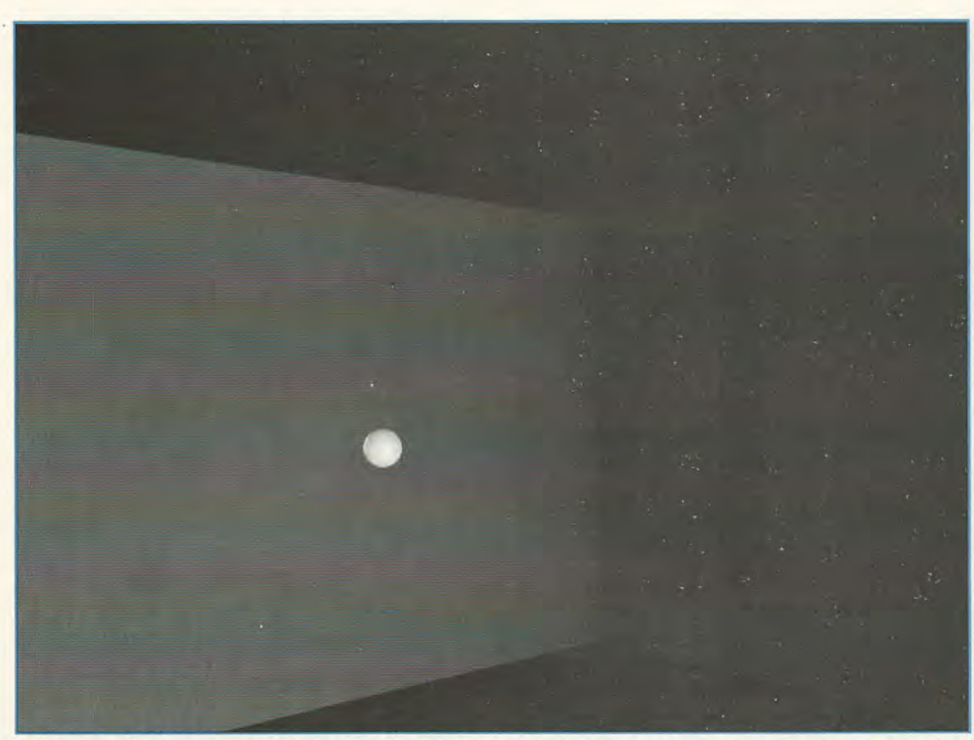


FIGURA 2. MUESTRA DEL MISMO RENDER DE LA FIGURA 1, PERO AHORA CON INTERPOLACION GOURAUD. LAS LIMITACIONES DE ESTA TÉCNICA IMPIDEN MOSTRAR BRILLOS EN EL INTERIOR DE LOS POLIGONOS. UNA SOLUCION POSIBLE ES AUMENTAR LA DENSIDAD POLIGONAL DE LAS ESCENAS.

¿Están empleado estos juegos sombreado *Phong*? Lógicamente la respuesta es no. Ya explicamos los costes del sombreado *Phong* y llegamos a la conclusión de que es imposible realizarlo en tiempo real (al menos con las máquinas actuales). Por otro lado, las tarjetas aceleradoras domésticas actuales sólo incorporan sombreado *Gouraud*. Entonces, ¿cómo se las apañan juegos como *Quake*? Hemos de introducir un nuevo concepto: mapas de luces.

LOS MAPAS DE LUCES

Los mapas de luces van asociados a los polígonos que forman cada una de las escenas de un programa. Así si nuestra escena tiene 1000 polígonos, necesitamos 1000 mapas de luces (como mínimo, porque como veremos un poco más abajo, un polígono puede tener más de un mapa de luz asociado). En el mapa de luz guardamos información sobre la cantidad de luz que cae en el polígono correspondiente.
Básicamente podemos considerar los mapas de luces como texturas. Veamos un ejemplo para aclarar conceptos.
En la Figura 3 tenemos una textura cualquiera, que no tiene un mapa de luz que se encuentre asociado a ella. De todas maneras, se trata de hecho de una textura que ha sido aplicada a un polígono determinado. Pero con la peculiaridad de que el mapa de luz posee información adicional que puede variar de *frame* a *frame*.

PRECÁLCULO DE LOS MAPAS DE LUCES

Podemos dividir los mapas de luces, en lo que toca al precálculo que se hace de ellos, en dos categorías distintas:

- Mapas de luces estáticos: aquellos que representan información que permanece constante a lo largo de un espacio de tiempo. No se ven alterados en ningún momento por el desarrollo del juego ni por otro tipo de agentes externos. Básicamente contribuyen a una creación de los escenarios mucho más realistas de lo que es habitual.
  El programa *Quake*, uno de los más conocidos arcades tridimensionales de este momento, es un excelente ejemplo de ello. Los precálculos de los mapas de luces suelen ser complicados. Se suele recurrir a técnicas de *Ray Tracing* (lanzar rayos desde la fuente de luz y comprobar si llegan al polígono o por el contrario chocan con algún objeto de la escena) o *Radiosity* (estudio de luz desde el punto de vista de la energía. Calculas la cantidad de energía que absorbe y emite cada superficie).

Los mapas de luces dinámicos nos dan la posibilidad de crear excelentes efectos visuales

- Mapas de luces dinámicos: son los mapas de luces temporales. Se crean como necesidad para algún evento concreto o bien varían a lo largo del tiempo. Supongamos que un personaje del juego lleva una antorcha en la mano. Si queremos que nuestro motor sea realista, entonces debemos calcular con todo detalle la iluminación de la antorcha en los polígonos que estén cercanos a ella.
  En la figura 4 tenemos un buen ejemplo de mapa de luz dinámica que ha sido generado a partir del lanzamiento de un arma (juego Quake).

Una vez que tenemos los mapas de luces calculados para un *frame* determinado (pueden ser más de uno para cada polígono) pasamos a la fase de aplicación.

FIGURA 3. TEXTURA DE EJEMPLO A LA QUE APLICAREMOS UN MAPA DE LUZ.



A vuestra disposición

Ya ha pasado un año desde que tomé las riendas de esta sección. 13 artículos en los que, supongo, todos hemos disfrutado. Yo el primero, he aprendido mucho con esta sección y sobre todo he aprendido mucho de los mails que he recibido, con el apoyo de la gente y la multitud de consejos e ideas nuevas.
Actualmente no dispongo del tiempo necesario para seguir haciendo esta sección. Así que me veo obligado a abandonarla (quién sabe si temporalmente) y dejarla en manos de un experto en estos mundos.
Mi dirección de correo electrónico sigue abierta para todos los que queráis seguir charlando conmigo con respecto a estos temas, para cualquier duda, etcétera.

icg_ent@bigfoot.com ***Jesús de Santos García Ent / Incognita***





FIGURA 4. IMAGEN DE QUAKE II. EN ESTA IMAGEN TENEMOS UN CLARO EJEMPLO DE MAPA DE LUZ DINAMICO ASOCIADO AL MISIL QUE SE ACABA DE LANZAR EN LA IMAGEN.

PINTADO DE LOS MAPAS DE LUCES

El pintado de los mapas de luces se reduce a la aplicación de técnicas de texturación. Suponemos que ya tenemos asignado a cada polígono sus mapas de luces específicos. Éstos están asignados con coordenadas <u,v> en cada uno de los vértices (los mapas de luces no son, de hecho, más que texturas). Con estos datos pintamos con multitextura cada uno de los polígonos teniendo en cuenta los siguientes apartados:

- Las últimas tarjetas 3D del mercado soportan multitexturas por hardware. Es decir, poseen varias unidades de texturación que permiten aplicar 2 o más texturas con una sola pasada. Generalmente el número suele ser 2. Si no disponemos de hardware para multitextura deberemos emular dicha capacidad pintando en varias pasadas sucesivas. A partir de la primera pasada debemos activar ALPHA_BLENDING para que los sucesivos mapas de luces se añadan a la textura principal.

Para pintar los mapas de luces hay que utilizar técnicas de texturización

Como alternativa a la multipasada, podemos fusionar previamente la textura con los mapas de luces y luego aplicar la textura con una única pasada. Esto tiene el inconveniente de que es bastante costoso para cada *frame* del motor. Para aliviar la carga se suelen emplear sistemas de caché: realmente los mapas de luces asociados a un polígono se suelen mantener a lo largo del tiempo (a no ser que aparezcan mapas de luces dinámicos temporales).

- El tamaño de los mapas de luces debe ser el más pequeño posible. Recordemos que existe un mapa de luz distinto por cada uno de los polígonos que tenemos. Debido a esto, el tamaño medio de los mapas de luces oscila entre 16x16 - 32x32, aunque esto ya depende de la implementación concreta. Con este tamaño es necesario aplicar filtros de interpolación a la hora de pintar los mapas de luces. Usualmente se recurre al filtro bilineal soportado por prácticamente todas las tarjetas medias actuales.

FIGURA 5. MAPA DE LUZ DIFUSA.

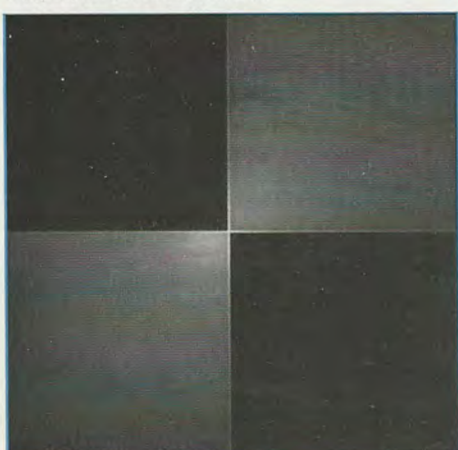
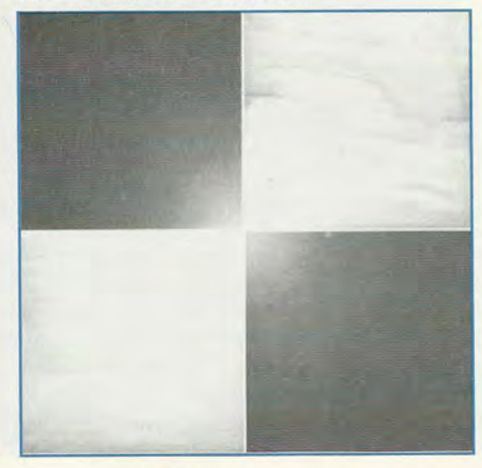


FIGURA 6. MAPA DE LUZ ESPECULAR.



Lo mapas de luces se pueden pintar en pantalla de forma difusa o especular

Podemos clasificar los mapas de luces con respecto al modo de pintarlos en pantalla:

- DIFUSOS: mapa de luz que indica el nivel de oscuridad en cada punto de la textura. Su misión es oscurecer la textura, manteniendo el brillo máximo al nivel original. Para estos mapas de luces se emplea ALPHA_BLENDING multiplicativo:

  $tR * lR = R$ (t = textura, l = mapa de luz)
  $tG * lG = G$
  $tB * lB = B$

- ESPECULARES : añaden brillo a cada punto de la textura. Simulan reflexión y brillo en superficies. Su misión es saturar la textura hasta un color especular. Para estos mapas de luces se emplea ALPHA_BLENDING aditivo:

  $tR + lR = R$
  $tG + lG = G$
  $tB + lB = B$

En las figuras 5 y 6 tenemos claramente diferenciados ambos tipos de mapa de luz.
Y esto es todo con respecto a los mapas de luces por el momento. Este mes no incluimos ningún código de ejemplo acompañando el contenido del CD-Rom de la revista. Se deja como tarea del lector la aplicación de mapas de luces al *engine* que llevamos construido hasta ahora. Para cualquier tipo de consulta, tenéis disponible como siempre la dirección del autor de estas líneas.

Autor: Pablo Trinidad

Primeros pasos con DIV

DIV es la herramienta perfecta para desarrollar juegos, tanto para aquellos que no tienen conocimientos de programación general, como para profesionales que buscan un sistema sólido de desarrollo. En esta serie de artículos aprenderemos de forma práctica cómo crear ese juego que siempre soñó.

El desarrollo de cualquier programa o juego comienza con el conocimiento profundo del entorno en el que se está trabajando. Por tanto, antes de comenzar con nuestro primer programa, analizaremos los puntos más importantes del entorno gráfico de DIV Games Studio.

EL ENTORNO GRAFICO DE DIV

DIV posee un sistema de menús en forma de ventanas, muy parecido al de Windows 95, por lo que a los usuarios de este sistema operativo les será mucho más fácil adaptarse al entorno. Analizaremos una a una las opciones básicas del menú principal.

DIV POSEE UN SISTEMA DE VENTANAS MUY SIMILAR AL DE WINDOWS 95

Programas: es el menú destinado al manejo de ventanas de programa, es decir, aquellas donde irá el código que va a definir el comportamiento de nuestro juego, así como el control de la ejecución, depurado e instalación de ellos. Éstas son las opciones básicas: *Nuevo*, *Abrir* (F4), *Cerrar*, *Guardar* (F2) y *Guardar como*: Gestiona el almacenamiento en disco y carga de los archivos de programa.

La interfaz que nos ofrece DIV es tan intuitiva y sencilla como la de Windows

Menú de Edición: Abre una ventana con las opciones de *Copiar*, *Mover*, *Borrar*, *Buscar*, *Marcar Bloques* y *Reemplazar* entre otras, con las que podemos modificar fácilmente nuestros programas.
Ejecutar (F10): Compila (crea el archivo ejecutable) y ejecuta el programa abierto.
Compilar (F11): Compila el programa seleccionado sin ejecutarlo.
Mapas: sus opciones están destinadas al manejo y creación de imágenes gráficas. Sus opciones son:
Opciones de archivo (*nuevo*, *abrir*...): Crea, carga y guarda los archivos .MAP (extensión asociada a los archivos de mapas). Además, pueden importarse archivos gráficos con formato PCX y BMP de 256 colores, que serán automáticamente grabados con extensión *.map*.
Reescalar: Amplía o reduce el tamaño de la imagen seleccionada. Se puede hacer de forma porcentual o expresando el nuevo tamaño en puntos. Además podemos convertir una imagen en color a escala de grises (incluso sin cambiar el tamaño).
Editar mapa: Abre el editor gráfico, con el que podremos modificar o crear cualquier imagen abierta. Se explicará detalladamente en próximos artículos.
Generador de explosiones: Crea una serie de imágenes de animación para explosiones. También será explicada más adelante.

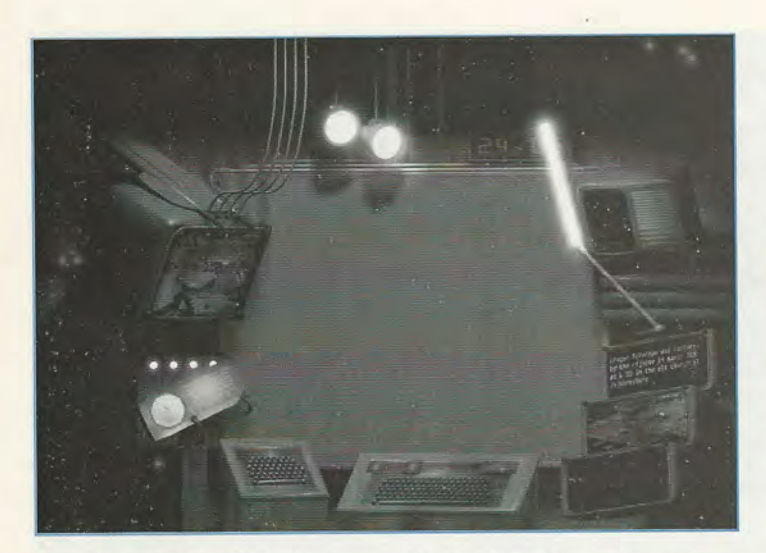
Imagen del entorno de DIV Games Studio con todas las opciones abiertas. Se pude observar su semejanza con el sistema de ventanas de Windows 95.



Ficheros: los ficheros son archivos de colecciones de imágenes con una paleta común. Su uso es sencillo. Basta con arrastrar cualquier ventana de mapa sobre la ventana de fichero para añadir imágenes. Con un simple clic sobre una imagen del fichero, se pueden seleccionar o deseleccionar éstos.

El lenguaje de programación de DIV es apto para todos los niveles de conocimiento

Posteriormente se pueden abrir todos en ventanas de mapas con la opción *Cargar Marcados*, o seleccionando la opción *Info* de la ventana de ficheros, obtener la información asociada a cada uno de los mapas (su tamaño, su centro y una imagen reducida de éste).
Sistema: consta de varias opciones interesantes para el control del entorno entre las que cabe destacar herramientas como un reproductor de CDs, un reloj, una papelera (para eliminar cualquier ventana de mapas o ficheros) y opciones tan interesantes como *Shell* a DOS o configuración, modo de vídeo y tapiz de fondo, con las que se podrá definir el entorno a su gusto.
Ayuda: hipertexto en el que podrá encontrar cualquier información acerca del entorno y el lenguaje de programación DIV.
Además de estas opciones hay otras muchas de importancia como el manejo de sonidos,



fuentes y paletas, así como el trazador de programas, una de las herramientas más potentes de este entorno, que iremos analizando poco a poco a lo largo del curso.

Estructura principal de un programa de DIV

PROGRAM ejemplo

BEGIN
<sentencias>;
END

Datos
No sólo de sentencias vive el programador. Todo programa necesita de unos datos o valores numéricos que se almacenen en memoria, como por ejemplo los puntos conseguidos en un matamarcianos o el resultado en un partido de fútbol. A estos valores se les denomina datos. Hay diversos tipos de datos, pero en este primer capítulo sólo estudiaremos las variables, por ser el más sencillo.
Una variable es un valor numérico al que se le asigna un nombre, con el que nos referiremos a este valor. Para dar valores a las variables se recurre a una sentencia tal como ésta:
Variable = <número>;
Donde «número» es un entero cuyo valor puede oscilar entre _2.147.483.648 (no se pueden utilizar decimales). Una vez asignado el valor a la variable, se puede utilizar a dicha variable para referirse al valor almacenado en ella. Por ejemplo:
Puntos = 500;
Total = puntos + 200;
Por tanto la variable «Total» almacenará el valor 700 (500+200).

Podemos distinguir 3 tipos de datos según su área de repercusión, es decir, en qué parte del programa son conocidos:
Datos globales: Se puede acceder a ellos en cualquier punto del programa. Se declaran (asigna valores) antes de las sentencias principales, es decir, entre *PROGRAM* y *BEGIN*. El comienzo de la declaración comienza con *GLOBAL* (Cuadro 2).
Datos locales: Es un dato que en realidad son varios, pero denominados igual y con valor distinto en cada proceso.
Datos privados: Sólo puede accederse a ellos dentro del proceso en el que son declarados.
Para entender bien estos dos últimos tipos de datos, es necesario conocer cuál es la estructura de un proceso.

No hemos definido todas las opciones a fondo porque el artículo se eternizaría. Por ello es aconsejable que exploremos por nuestra cuenta todos los entresijos de las opciones anteriormente descritas. No obstante, algunas de ellas irán apareciendo en sucesivos artículos, a medida que se necesiten.

EL LENGUAJE DIV

Otro aspecto necesario para desarrollar juegos con DIV es conocer su lenguaje de programación. Al respecto realizaremos una breve introducción e indicaremos algunos aspectos esenciales para que podamos comenzar con el desarrollo de nuestro primer ejemplo lo antes posible.

ESTRUCTURA GENERAL

Un programa es una sucesión de órdenes, que el ordenador interpreta una por una y que, por tanto, determinarán cuál debe ser el comportamiento de nuestro juego. En un programa de DIV encontramos 3 partes diferenciadas:

Un proceso es todo objeto o gráfico que existe dentro de un juego

- Una zona de definición de datos.
- Las órdenes que debe ejecutar nuestro juego
- Las órdenes que se deben ejecutar con cada proceso.

Entendemos como proceso a todo objeto dibujado en la pantalla y que se puede mover a través de ella. Es un proceso, por ejemplo, un coche en un juego de carreras.
En DIV, cada programa comienza con la sentencia *PROGRAM <archivo>*, donde archivo indica el nombre del programa en cuestión sin la extensión *.prg*. A continuación, encontraremos dos sentencias, entre las cuales escribiremos las órdenes que el juego debe ejecutar: *BEGIN* y *END*. Por tanto, contamos con un esqueleto muy básico común a todos los programas. En el cuadro 1 se ilustra un ejemplo de la estructura más sencilla que puede tener un programa en DIV.

EJEMPLOS DE SENTENCIAS USANDO OPERACIONES CON VARIABLES

```
PROGRAM ejemplo2

GLOBAL
Puntos = 0;
Total = 20;

BEGIN
Puntos = Puntos + 40;
Total = Puntos - Total;
END
```

PROCESOS

Como ya dijimos anteriormente, un proceso es todo objeto o gráfico que existe en un juego. Para definir el comportamiento de cada proceso, se crean unos bloques de sentencias. Cada uno de estos bloques representa a un tipo de proceso, es decir, que si queremos dibujar en la pantalla 10 coches iguales, basta con crear 10 procesos del mismo tipo y que dibujen un coche, con lo que tan sólo sería necesario un bloque de sentencias. Esto se verá más claro en nuestro programa de ejemplo. Para definir un bloque de sentencias se recurre a la estructura siguiente:

```
PROCESS <nombre del proceso> ( )
PRIVATE
<declaración de datos>;
...
BEGIN

<sentencias>;
...
END
```

Notamos la presencia de una sección *PRIVATE*, donde se declararán los datos privados, es decir, aquellos a los que sólo el proceso va a tener acceso y, por tanto, serán inaccesibles para el resto del programa. Los procesos son objetos en pantalla con un código y variables asociados



LA HORA DE LA VERDAD: NUESTRO PRIMER PROGRAMA

Una vez realizada esta introducción teórica, es hora de comenzar con la práctica. Vamos a realizar una sencilla animación, con lo que iremos aprendiendo las distintas funciones del lenguaje DIV según se vayan necesitando. Si no se tienen conocimientos previos de programación, tal vez le resulte un tanto escabroso. No hay problema si no entiende todo, con los ejemplos se irán asimilando los conceptos.

El programa que vamos a realizar es un simple desplazamiento de un coche por la pantalla. Para ello, en primer lugar es necesario tener las imágenes de coche. Podemos dibujarlo nosotros con el editor gráfico incorporado o cargar una imagen prediseñada de la librería de DIV Games Studio. Es preferible esta última opción, ya que aún no sabemos manejar el editor gráfico. Éstos son los pasos a seguir para obtener los gráficos:

Abrimos un archivo de mapas de coches, por ejemplo uno situado en el directorio *\Librería\coches*.

Es imprescindible ponerse a programar para dominar plenamente el lenguaje

Nosotros hemos elegido el archivo *coches1.map*. Para ello hemos seleccionado la opción *Abrir* del menú de mapas.

Nos aparecerá el mapa elegido abierto en una ventana. Hacemos doble clic sobre el fondo de la imagen para que ésta se abra con el editor gráfico.

Una vez en el editor gráfico, seleccionamos la opción de edición de bloques (simbolizado por unas tijeras y un recuadro). Seleccionamos la imagen deseada haciendo clic dos veces, uno para comenzar la selección y otro para terminarla.

Acto seguido, pulsamos en la nueva opción que aparece en la barra de opciones simbolizada con una ventana. Salimos del editor pulsando el botón derecho del ratón.Con esto, habremos creado un nuevo mapa con las dimensiones de la selección.

Si sigues todos los pasos con cuidado llegarás a realizar un juego en poco tiempo

Creamos un nuevo fichero, de nombre *ejemplo1.fpg* con la opción nuevo del menú *Ficheros*. Arrastramos la ventana del nuevo mapa sobre la ventana del fichero (pulsando sobre la imagen y no sobre el título). Aparecerá una nueva ventana donde nos pedirá el código del mapa (elegiremos el 001) y una descripción del mapa



RESULTADO DEL CAMBIO DE IMAGEN POR MEDIO DE LA PULSACION DEL CURSOR. CON ESTO SE OBTIENE UN REALISMO MUCHO MAYOR EN EL EFECTO DE MOVIMIENTO, MIRANDO HACIA DONDE SE MUEVE EL VEHICULO.

MODOS DE VIDEO DISPONIBLES EN DIV. TODOS TIENEN 256 COLORES

VGA estándar	M320x200
Modo X	M320x240
	M320x400
	M360x240
	M360x360
	M376x282
SVGA VESA	M640x400
	M640x480
	M800x600
	M1024x768



Load_Fpg(<fichero>): Carga el fichero de imágenes que va a utilizar el programa. Se puede especificar una ruta para encontrar el archivo. Si no es así, lo buscará en el directorio por defecto (*.\FPG*).

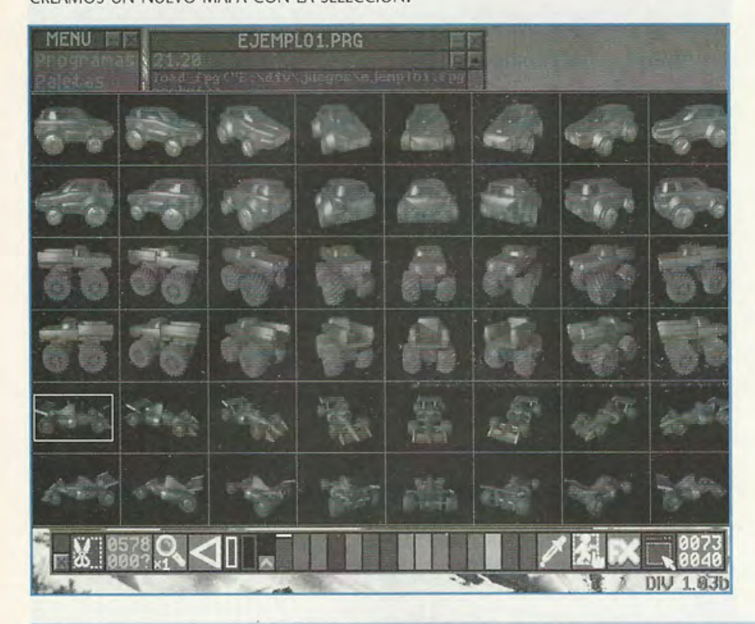
FRAME: Podríamos decir que es la orden de visualización. Indica cuándo un proceso está listo para ser visualizado. Cuando todos los procesos estén listos, la siguiente imagen se dibujará en pantalla. En el código principal también debe indicarse cuándo estamos listos para la siguiente visualización.



LOOP...END: Es lo que denominamos un bucle, es decir, una repetición de sentencias. Todas las sentencias situadas entre estos 2 comandos, se repetirán infinitamente. Se suele utilizar junto con la sentencia FRAME para que la muestra de imágenes tenga continuidad.

Coche( ): Crea un proceso del tipo coche, definido más abajo.

UNA VEZ REALIZADA LA SELECCION DE LA IMAGEN DESEADA, PULSAMOS LA OPCION DE CORTAR EN VENTANA, SITUADA EN LA ESQUINA INFERIOR DERECHA. CON ESTO CREAMOS UN NUEVO MAPA CON LA SELECCION.



(el que elijamos).
Una vez realizados estos pasos, podemos empezar a escribir nuestro código. Cremos un nuevo programa llamado *ejemplo1.prg*. En primer lugar vamos simplemente a colocar la imagen del coche en la pantalla. Para ello, vamos a crear un proceso llamado *coche()*. El código sería el siguiente:

```
PROGRAM ejemplo1;
BEGIN

set_mode(m640x480);
load_fpg("ejemplo1.fpg");
coche( );
LOOP
FRAME;
END
END
PROCESS coche()
BEGIN
graph = 1;
x =100;
y=100;
LOOP
FRAME;
END
END
```

Vamos a analizar las distintas sentencias que aparecen en el programa:
Set_Mode(<modo>): inicia el modo de vídeo que deseamos para el juego. Los valores posibles aparecen en el Cuadro 3. Si no se especifica esta sentencia al iniciar el programa, automáticamente se asigna el modo de vídeo 320x200. En nuestro juego utilizaremos el modo SVGA VESA de 640x480. Todos los modos tienen 256 colores.

PRINCIPALES CODIGOS DE TECLADO DISPONIBLES EN DIV

_esc:	Escape
_f?:	Tecla de función F? (?=1..12)
_tab:	Tabulador
_enter:	Enter
_l_shift	Mayúsculas izquierdo
_r_shift	Mayúsculas derecho
_control	Teclas control
_alt	Alt o Alt Gr
_space	Barra espaciadora
_0..._9:	Números del 0 al 9
_a..._z:	Letras de a...z
_up:	Cursor arriba
_down:	Cursor abajo
_left:	Cursor izquierda
_right:	Cursor derecha
_backspace:	Borrado <=
_ins:	Insertar
_del:	Suprimir

Teniendo en cuenta lo anteriormente descrito, podemos conocer si una tecla, en este caso el cursor derecho, está pulsada mediante una sentencia *IF* de la forma:

```
IF (key(_right)==1)
x=x+1;
END
```

Si está pulsada, incrementa el valor de x. También podemos abreviar la expresión, atendiendo al valor devuelto por *key()*. Si está pulsada devuelve 1, es decir, un valor impar, y si devuelve 0, un valor par. Por tanto, aplicándolo al último caso estudiado de la sentencia IF, podemos abreviar de la siguiente forma:

```
IF(key(_right))
x=x+1;
END
```

Por tanto, ya podemos dotar a nuestro programa de control total con las teclas de cursor:

```
PROGRAM ejemplo1;

BEGIN
set_mode(m640x480); load_fpg("ejemplo1.fpg");
coche( );
LOOP
FRAME;
END
END
PROCESS coche()
BEGIN
graph = 1;
x =100;
y=100;
LOOP
IF (key(_right))
x=x+1;
END
IF(key(_left))
x=x-1:
END
FRAME;
END
END
```

No pierdas de vista las opciones básicas y los códigos de teclado que posee DIV

Es evidente que también podemos realizar otras muchas acciones con lo aprendido hasta ahora. Por ejemplo, si repetimos el proceso para crear el gráfico del coche, y creamos otro coche mirando hacia el lado contrario, y le asignamos el código 002, podemos cambiar la imagen del coche según la tecla pulsada con la sentencia:
Graph=2;
A lo largo de este artículo hemos realizado una introducción rápida, en la que hemos visto las opciones básicas de movimiento de imágenes. No obstante esto es tan sólo el principio. Experimentaremos con lo aprendido hasta ahora: cambiaremos el modo de vídeo, las teclas a pulsar, incrementaremos el desplazamiento en más unidades,etcétera. Son sorprendentes las amplias posibilidades de DIV.En el próximo artículo veremos cómo añadir fondos a nuestros juegos y cómo realizar animaciones de forma sencilla.

Autor: José Antonio Guerra
balder@mindless.com

Sonido en DirectX [II]

¿Cómo? ¿Qué aún tenéis la miel en los labios tras empezar a tratar con el sonido? Pues este mes traemos todo para calmar vuestra sed de conocimientos. Entremos en profundidad con el manejo de sonidos mediante DirectSound, justo donde lo dejamos la entrega pasada.

Para una aplicación que simplemente va a reproducir sonidos a través del dispositivo por defecto, como en el ejemplo del mes pasado, no es necesario efectuar una enumeración de los dispositivos. Vimos que cuando creábamos un objeto DirectSound pasándole NULL como identificador de dispositivo, la interfaz se asociaba automáticamente al dispositivo por defecto. Si no había dispositivos, la llamada a *DirectSoundCreate* fallaba.

De todas formas, si lo que queremos es encontrar un tipo particular de dispositivo (alguno que nos dé soporte para sonido 3D, por ejemplo) o necesitamos trabajar con dos o más dispositivos, debemos hacer que DirectSound enumere todos los dispositivos de sonido que se encuentren en el sistema.

La enumeración sirve para tres propósitos:
Informa de los dispositivos disponibles.
Nos entrega un GUID para cada dispositivo.
Nos permite iniciar DirectSound con cada dispositivo conforme va siendo enumerado.

Para trabajar con varios dispositivos, debemos hacer que DirectSound enumere los dispositivos de sonido del sistema

Para enumerar los dispositivos debemos establecer una función *callback* que será llamada cada vez que DirectSound encuentre un dispositivo. Podemos hacer todo lo que queramos con esta función, y podemos llamarla como queramos, pero debe ser declarada de la misma forma que *DSEnumCallback* (ver recuadro).

La función *callback* debe retornar TRUE si la enumeración debe seguir, o FALSE si queremos parar dicha enumeración de los dispositivos (por ejemplo, cuando hayamos encontrado un dispositivo con las capacidades que andábamos buscando).

Si queremos trabajar con uno o más dispositivos - por ejemplo, uno de captura y otro de reproducción - la función *callback* es un lugar idóneo para crear e iniciar el objeto *DirectSound* para cada dispositivo.

El siguiente ejemplo, extraído de Dsenum.c (ejemplo Dsshow del SDK), enumera los dispositivos existentes y añade información acerca de cada uno en una lista de un control *combo box*. Veamos la función *callback* íntegra:

```
BOOL CALLBACK DSEnumProc(LPGUID lpGUID,
                 LPCTSTR lpszDesc,
                 LPCTSTR lpszDrvName,
                 LPVOID lpContext )
  {
  HWND   hCombo = *(HWND *)lpContext;
  LPGUID lpTemp = NULL;

  if( lpGUID != NULL )
      {
      if(( lpTemp = LocalAlloc( LPTR,
sizeof(GUID))) == NULL )
      return( TRUE );

      memcpy( lpTemp, lpGUID, sizeof(GUID));
  }

  ComboBox_AddString( hCombo, lpszDesc );
  ComboBox_SetItemData( hCombo,
          ComboBox_FindString( hCombo, 0,
lpszDesc ),
          lpTemp );
  return( TRUE );
  }
```

La enumeración comienza cuando el diálogo que contiene el control *combo box* es iniciado:

```
if
(DirectSoundEnumerate((LPDSENUMCALLBACK)
DSEnumProc, &hCombo)
  != DS_OK )
  {
  EndDialog( hDlg, TRUE );
  return( TRUE );
  }
```

ESQUEMA DE LA ORGANIZACION DE LOS SONIDOS EN MEMORIA.

Organizacion de los sonidos estéreo

Canal I
Canal D

Las muestras en memoria

I	D	I	D	I	D	I	D	I	D

Prototipo de DSEnumCallback

```
BOOL CALLBACK DSEnumCallback(LPGUID lpGUID,
LPCTSTR lpszDesc,
LPCTSTR lpszDrvName,
LPVOID lpContext )
```



Opciones de control de los buffers de sonido

Al crear un buffer de sonido, nuestra aplicación debe especificar las opciones de control necesarias para dicho buffer. Esto se hace a través del miembro dwFlags de la estructura DSBUFFERDESC, la cual puede contener uno o más de los flags DSBCAPS_CTRL*. DirectSound utiliza esas opciones cuando reserva recursos hardware para los buffers de sonido. Por ejemplo, un dispositivo puede soportar buffers en hardware, pero no verse provisto del control de balance de canales en dichos buffers. En este caso, DirectSound usaría aceleración hardware sólo si el flag DSBCAPS_CTRLPAN no fuese especificado.
Para obtener el mejor rendimiento en todas las tarjetas de sonido, nuestra aplicación debe especificar únicamente las opciones de control que vaya a utilizar.
Si nuestra aplicación llamase a un método que el buffer no soporte, dicho método fallará. Por ejemplo, si intentamos cambiar el volumen usando la función IDirectSoundBuffer::SetVolume, el método funcionará si el flag DSBCAPS_CTRLVOLUME fue especificado en la creación de dicho buffer. De otra manera, el método fallará y retornará el error DSERR_CONTROLUNAVAIL. El proveer de controles a los buffers ayuda a asegurar que todas las aplicaciones funcionarán correctamente en dispositivos actuales y futuros.

Cabe notar que la dirección del *handle* del *combo box* es pasada en la función *DirectSoundEnumerate,* la cual a su vez la pasa a la función *callback.* Este parámetro puede ser cualquier valor de 32 bits al que queramos tener acceso desde dentro del *callback.*

SEGUIMOS CON LOS BUFFERS 1DE SONIDO

El mes pasado explicamos los fundamentos básicos de los *buffers* de sonido y dimos un ejemplo práctico de cómo crearlos. Como apenas nos dio tiempo a meternos en profundidad, pasemos a ver todo lo necesario para dominarlos.
Recordemos que al iniciar el objeto *DirectSound,* automáticamente se creaba un *buffer* de sonido primario para mezclar los sonidos y mandarlos al dispositivo de salida. En nuestras aplicaciones debemos crear al menos un *buffer* secundario para almacenar y reproducir sonidos individuales. Un poco más adelante profundizaremos en la creación y manejo de estos *buffers.*

Un *buffer* secundario puede existir durante toda la vida de la aplicación o puede ser destruido cuando ya no sea necesario. Puede contener un sólo sonido para ser reproducido repetidamente, tal como un efecto de sonido de un videojuego, o puede ser llenado con nuevos datos cada vez. La aplicación puede reproducir un sonido almacenado en un *buffer* secundario como un solo evento o como un sonido cíclico que se reproduce constantemente.

Hay que tener en cuenta numerosos factores para llegar a dominar los *buffers* de sonido

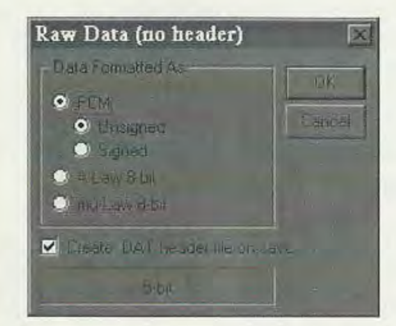
Se pueden crear dos o más *buffers* secundarios en la misma memoria física usando la función *IDirectSound::DuplicateSoundBuffer,* contando con que el *buffer* original no esté en la memoria del dispositivo.
Se pueden mezclar sonidos desde *buffers* secundarios distintos simplemente reproduciéndolos a la misma vez. Los datos de los *buffers* secundarios son mezclados por DirectSound en el *buffer* primario.
No existe límite en el número de *buffers* secundarios que se pueden reproducir a la vez, salvo los que establezca el hardware (en caso de ser muestras en la memoria del dispositivo) y la capacidad de procesamiento de la CPU.
El mezclador de *DirectSound* puede ofrecer una pequeña latencia (el tiempo que transcurre desde que se da la orden de reproducción hasta que el sonido es reproducido por el dispositivo) de tan sólo 20 milisegundos, por lo tanto no hay retardo apreciable hasta que la reproducción comienza. Bajo estas condiciones, si nuestra aplicación reproduce un *buffer* e inmediatamente iniciamos una animación gráfica, el sonido y la imagen aparentan comenzar al mismo tiempo. Por otra parte, si *DirectSound* debe emular características hardware en software, el mezclador no puede lograr una baja latencia y un retardo mayor (típicamente 100-150 milisegundos) transcurren antes de que se reproduzca el sonido (en una aplicación que transcurra a una velocidad de refresco de 30 fotogramas por segundo, esto supone unos 3 o 4 fotogramas de latencia).

CREACION DE BUFFERS SECUNDARIOS

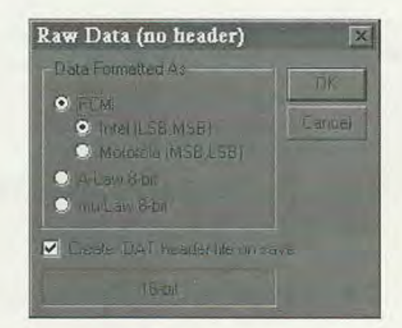
Retomemos el final de la entrega pasada y pasemos a señalar más aspectos de la creación de *buffers* de sonido secundarios.
Nuestras aplicaciones deberían crear *buffers* en primer lugar para los sonidos más importantes, y entonces crear *buffers* para otros sonidos en un orden descendente de importancia.
DirectSound reserva recursos de hardware para los primeros *buffers* que puedan aprovecharse de ellos.
Si nuestra aplicación debe explícitamente crear *buffers* de sonido en hardware o software, podemos especificar el *flag DSBCAPS_LOCHARDWARE* o el *flag DSBCAPS_LOCSOFTWARE* en la estructura *DSBUFFERDESC.* Si el *flag DSBCAPS_LOCHARDWARE* es especificado y no hay suficiente memoria o capacidad de mezcla, la creación del *buffer* falla.
Podemos averiguar la localización de un *buffer* de sonido existente usando el método *IDirectSoundBuffer::GetCaps* y comprobando el valor del miembro *dwFlags* de la estructura *DSBCAPS* en busca de los valores *DSBCAPS_LOCHARDWARE* o *DSBCAPS_LOCSOFTWARE.* Una de las dos está siempre especificada.
Los objetos *DirectSoundBuffer* son propiedad del objeto *DirectSound* que los creó. Cuando dicho objeto *DirectSound* es liberado, todos los *buffers* creados por él serán también liberados y no deben ser usados más.

FORMATO CORRECTO DE FICHEROS RAW.

Opciones de ficheros RAW



8-bit Unsigned



16-bit Signed



Grabando ficheros RAW con Cool Edit 96.

REPRODUCCION DE SONIDO

La reproducción del sonido consta de los siguientes pasos:

- Bloquear una porción del *buffer* de sonido secundario (*IDirectSoundBuffer::Lock*). Este método devuelve un puntero a la dirección donde comenzará la escritura, basado en el desplazamiento desde el comienzo del *buffer* que nosotros le pasemos.
- Escribir los datos del sonido al *buffer*.
- Desbloquear el *buffer* (*IDirectSoundBuffer::Unlock*).
- Mandar el sonido al *buffer* primario y desde allí al dispositivo de salida (*IDirectSoundBuffer::Play*).

Dado que los *buffers* de sonido que se reproducen cíclicamente son en concepto circulares, DirectSound devuelve dos punteros de escritura cuando bloqueamos un *buffer* de sonido. Por ejemplo, si intentamos bloquear 300 bytes comenzando por el punto medio de un *buffer* de 400 bytes de largo, el método *Lock* devolverá un puntero a los últimos 200 bytes del *buffer*, y un segundo puntero a los primeros 100 bytes. El segundo puntero es NULL si la porción bloqueada del *buffer* no se solapa.

Le reproducción de sonido debe desbloquear una porción del *buffer* de sonido secundario

Normalmente, el *buffer* para su reproducción automáticamente cuando llega a su fin. En otro caso, si el *flag DSBCAPS_LOOPING* fue establecido en el parámetro *dwFlags* del método ***Play***, el *buffer* será reproducido repetidamente hasta que la aplicación llame al método *IDirectSoundBuffer::Stop*, punto en el que la posición de reproducción es movida al comienzo del *buffer*.

El siguiente ejemplo escribe datos a un *buffer* de sonido, empezando en el desplazamiento pasado en el parámetro *dwOffset*:

```
BOOL AppWriteDataToBuffer(
   LPDIRECTSOUNDBUFFER lpDsb,  // el buffer de sonido
   DWORD dwOffset,             // desplazamiento inicial
   LPBYTE lpbSoundData,        // puntero a nuestros datos
   DWORD dwSoundBytes)         // tamaño del bloque a copiar
{
   LPVOID lpvPtr1;
   DWORD dwBytes1;
   LPVOID lpvPtr2;
   DWORD dwBytes2;
   HRESULT hr;
   // Obtenemos la dirección de memoria del bloque de escritura.
   // Estará en dos partes si el bloque se solapa.
   hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes, &lpvPtr1,
      &dwBytes1, &lpvPtr2, &dwBytes2, 0);

   // Si DSERR_BUFFERLOST is es devuelto, restauramos y volvemos a intentar bloquear.
   if(DSERR_BUFFERLOST == hr) {
      lpDsb->lpVtbl->Restore(lpDsb);
      hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes,
         &lpvPtr1, &dwAudio1, &lpvPtr2, &dwAudio2, 0);
   }
   if(DS_OK == hr) {
      // Escribimos a los punteros.
      CopyMemory(lpvPtr1, lpbSoundData, dwBytes1);
      if(NULL != lpvPtr2) {
         CopyMemory(lpvPtr2, lpbSoundData+dwBytes1, dwBytes2);
      }
      // Desbloqueamos el buffer.
      hr = lpDsb->lpVtbl->Unlock(lpDsb, lpvPtr1, dwBytes1, lpvPtr2,
         dwBytes2);
      if(DS_OK == hr) {
         // Hemos tenido éxito.
         return TRUE;
      }
   }
   // Lock, Unlock, o Restore han fallado.
   return FALSE;
}
```

CONTROLES DE REPRODUCCION

Para obtener y establecer el volumen de reproducción de un *buffer*, podemos usar los métodos *IDirectSoundBuffer::GetVolume* y *IDirectSoundBuffer::SetVolume*. Si establecemos el volumen del *buffer* primario, cambiaremos el volumen de salida del dispositivo. Similarmente, llamando a los métodos

Gestión de Buffers

Podemos usar el método IDirectSoundBuffer::GetStatus para determinar si el buffer de sonido se está reproduciendo o está parado.
Podemos utilizar el método IDirectSoundBuffer::GetFormat para obtener información acerca del formato de onda del buffer. También podemos usar los métodos IDirectSoundBuffer::GetFormat y IDirectSoundBuffer::SetFormat para consultar y establecer el formato de onda del buffer primario. Cuando un buffer de sonido secundario es creado, su formato es fijo. Si necesitásemos un buffer secundario que use otro formato, debemos crear otro nuevo.
La memoria de un buffer de sonido puede perderse en determinadas situaciones. En particular, esto puede ocurrir cuando están situados en la memoria del dispositivo. En el peor de los casos, la misma tarjeta de sonido puede quitarse del sistema mientras se está usando, lo cual sólo es posible en tarjetas de sonido PCMCIA.
La pérdida puede ocurrir también cuando una aplicación con el nivel de cooperatividad write-primary es activada. En este caso, DirectSound marca todos los demás buffers como perdidos para que dicha aplicación pueda escribir directamente al buffer de sonido. El código de error DSERR_BUFFERLOST es devuelto cuando los métodos IDirectSoundBuffer::Lock o IDirectSoundBuffer::Play son llamados para cualquier otro buffer. Cuando la aplicación establece cualquier otro nivel de cooperatividad que no sea write-primary, o se pone en segundo plano, otras aplicaciones pueden entonces intentar volver a reservar la memoria del buffer mediante una llamada al método IDirectSoundBuffer::Restore. Si tiene éxito, este método restaura la memoria del buffer y todos los demás parámetros, tales como el volumen y el balance de canales. De todas formas, un buffer restaurado no contiene datos válidos de sonido. La aplicación debe reescribir los datos al buffer restaurado.

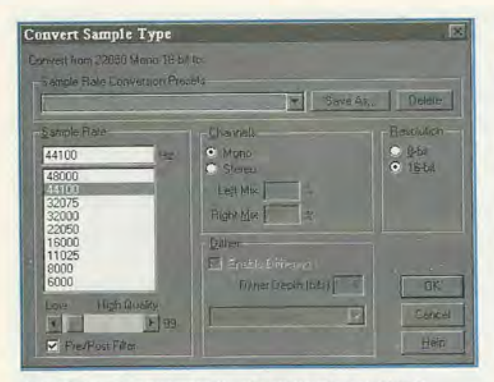
IDirectSoundBuffer::GetFrequency y *IDirectSoundBuffer::SetFrequency*, podemos consultar y cambiar la frecuencia a la cual se reproducen los sonidos. Eso sí, no podremos cambiar la frecuencia del *buffer* primario.

Para establecer y consultar el balance de canales, podemos llamar a los métodos *IDirectSoundBuffer::GetPan* y *IDirectSoundBuffer::SetPan*. Tampoco podremos cambiar el balance del *buffer* primario.

POSICIONES DE REPRODUCCION Y ESCRITURA

DirectSound mantiene dos punteros en el *buffer*: la posición de reproducción actual y la posición de escritura actual. Estas posiciones son desplazamientos en bytes dentro del *buffer*, no direcciones absolutas de memoria. El método *IDirectSoundBuffer::Play* siempre comienza a reproducir desde la posición de





DIALOGO DE CONVERSION DEL TIPO DE MUESTREO.

reproducción actual del *buffer*. Cuando un *buffer* es creado, la posición de reproducción es situada al comienzo del mismo. Conforme el sonido se va reproduciendo, la posición de reproducción se mueve y siempre apunta al siguiente byte de datos. Cuando el *buffer* es parado, la posición de reproducción se queda donde esté.
La posición actual de escritura es el punto tras el cual es seguro escribir datos en el *buffer*. El bloque situado entre la posición de reproducción y la de escritura está destinado a ser reproducido, y por lo tanto no puede ser modificado con seguridad.
Imaginemos el *buffer* como la esfera de un reloj, con los datos escribiéndose en el sentido de las agujas. La posición de reproducción y la posición de escritura son como dos manecillas recorriendo la esfera a la misma velocidad, con la posición de escritura ligeramente delante de la posición de reproducción. Si la posición de reproducción apunta al 1 y la posición de escritura apunta al 2, solamente es seguro escribir datos tras el 2. Los datos entre el 1 y el 2 puede que estén esperando en la cola de reproducción de DirectSound y no deben ser tocados.
Una aplicación puede obtener las posiciones de escritura y reproducción actuales mediante una llamada al método *IDirectSoundBuffer::GetCurrentPosition*. El método *IDirectSoundBuffer::SetCurrentPosition* permite establecer la posición de reproducción, pero la posición de escritura no puede ser cambiada.

MEZCLANDO SONIDOS

Es fácil mezclar distintos sonidos con DirectSound. Simplemente hemos de crear *buffers* de sonido secundarios, recibiendo una interfaz *IDirectSoundBuffer* por cada sonido. Entonces, sólo hay que reproducir los *buffers* simultáneamente. DirectSound se hace cargo de la mezcla en el *buffer* primario de sonido y reproduce el resultado.
El mezclador de DirectSound puede obtener los mejores resultados de la aceleración hardware si la aplicación especifica correctamente el *flag* DSBCAPS_STATIC para los *buffers* estáticos (que no cambien su contenido habitualmente). Este *flag* debe ser especificado para cualquier *buffer* estático que vaya a ser reutilizado. DirectSound descarga estos *buffers* a la memoria de la tarjeta de sonido, cuando es permitido, y en consecuencia elimina de carga al procesador ya que no tiene que mezclar los *buffers* a mano.
Los *buffers* estáticos más importantes deberían ser creados en primer lugar para darles prioridad y aprovechar la aceleración por hardware.
El mezclador de DirectSound producirá la mejor calidad del sonido si todos los sonidos que usemos en nuestra aplicación usan el mismo formato de onda que el *buffer* primario. Si esto es así, el mezclador no debe entonces realizar ninguna conversión de formatos.

Para manejar ficheros de sonido tipo .wav de Windows se puede utilizar el programa Cool Edit

Podemos cambiar el formato de onda del dispositivo creando un *buffer* primario y llamando al método *IDirectSoundBuffer::SetFormat*. Hay que destacar que este *buffer* primario es creado solamente para propósitos de control, no para escribir en él. Es necesario estar como mínimo en un nivel de cooperatividad *DDSCL_PRIORITY* para llamar al método *SetFormat*. DirectSound restaurará el formato de onda del hardware al formato especificado en la última llamada cada vez que la aplicación vuelva al primer plano.

USO BASICO DEL COOL EDIT

Como comentamos el mes pasado, DirectSound no posee funciones para manejar ficheros de sonido, tales como los de tipo WAV de Windows. En el ejemplo vimos cómo cargar un sonido sin formato (RAW), que generamos con el editor de sonido Cool Edit 96.
Cool Edit es un programa muy completo para la edición de sonidos, si bien nosotros solamente lo vamos a utilizar para convertir sonidos de numerosos formatos a ficheros RAW, que manejaremos más fácilmente.
En el menú *File* – que será el único que usemos – nos encontramos con tres opciones que nos servirán para cumplir nuestros objetivos.
Vamos a verlas en detalle:

- *Open:* nos permite cargar un fichero de sonido en el editor.
- *Open As...:* igual que la anterior, pero en cambio nos permite variar el formato de onda mediante el cual será interpretado el fichero.
- *Save As:* nos permite grabar en el disco duro el sonido editado. Desde esta opción podremos establecer el formato del fichero resultante.

En resumen, el proceso de conversión es bastante simple; sólo debemos cargar el fichero origen mediante *Open* y grabarlo en el formato que deseemos mediante el comando *Save As*.
Si además queremos cambiar el formato de la onda, por ejemplo, convertir un sonido estéreo en uno mono, iremos al menú *Edit* y pincharemos en la opción *Convert Sample Type*. Desde el diálogo que nos aparecerá tendremos un control total del formato de onda del sonido.

CONSIDERACIONES SOBRE FICHEROS RAW

Los ficheros RAW pueden almacenar los datos del sonido de maneras muy distintas. Como son una imagen exacta de los datos que introduciremos en los *buffers* de sonido, hemos de saber cómo están organizados dichos datos para efectuar bien las conversiones.
Veamos la organización según el número de bits por muestra:

- **8 bits por muestra:** Cada muestra del sonido ocupa un byte, y es sin signo. Es decir, el valor de cada muestra oscila entre un mínimo de 0 y un máximo de 255. Es equivalente al tipo *unsigned char* del lenguaje C.
- **16 bits por muestra:** Cada muestra ocupa dos bytes, y es con signo. El mínimo tiene un valor de –32768 y el máximo es de 32768. Es equivalente al tipo *short* del lenguaje C. Cabe indicar que el formato de la palabra es el de Intel, es decir, el byte menos significativo se sitúa primero.

Si un sonido es estéreo, las muestras de cada canal se van almacenando entrelazadas, tal y como vemos en la figura.

EL MENU FILE DE COOL EDIT 96.



Despedida y cierre

En estos dos artículos dedicados al sonido hemos visto todo lo necesario para dotar de sonido a nuestros programas y juegos usando DirectSound. Esperamos que os sean de utilidad y que disfrutéis de la potencia de esta arquitectura para añadir espectacularidad a vuestras creaciones. ¡Hasta el mes que viene!

Autor: Carlos Gzlez. Morcillo

Curso de animación [III]

La tarea de construir un personaje no se resume únicamente en dibujar la figura. Cada personaje tiene su propia personalidad, sus formas y proporciones. En esta entrega del curso haremos hincapié en las técnicas más usadas para hacer modelos anatómicos correctos y realizaremos nuestra primera animación en Autodesk Animator Pro.

¿Es que no vas a dejar este trabajo? La gente habla mal de tus experimentos, muchos te rechazan porque te dedicas a esa ciencia infernal. ¿Por qué no abandonas?, seguro que serías feliz... - le comentaba un amigo a Andrés Vesalio mientras caminaban por su laboratorio. Los cartílagos esparcidos por el suelo, al ser pisados, producían extraños crujidos. Andrés hizo un gesto con la mano indicándole a su amigo que se callara. En medio del silencio se dio la vuelta, sonrió y le dijo a su amigo - ¿quieres hacerme feliz?... ¡regálame un cadáver humano!

Tras esta pequeña introducción que parece sacada de una película de terror, se esconde la historia del "padre" de la anatomía que conocemos hoy en día. Andrés Vesalio (o Andreas Vesalius) fue un anatómico belga que escribió a mediados del siglo XVI el libro *De humani corporis fabrica libri septem*, el primer tratado de anatomía humana exacto y completo. Desde estas páginas queremos rendirle nuestro pequeño homenaje.

PERSONAJES QUE SEAN ANATÓMICAMENTE CORRECTOS

¿Necesitamos conocer a la perfección la anatomía humana para hacer animaciones destinadas a un videojuego? Si bien es necesario tener algunas bases sobre anatomía, no tenemos que ser expertos científicos en esa materia. Básicamente debemos conocer las articulaciones y la situación de los músculos más grandes del organismo. Bastará con observar el movimiento que queremos animar unas cuantas veces para captar la esencia del mismo y trasladarlo en unos pocos *frames* en nuestra mesa de luz. Como ya comentamos en la primera entrega del curso, una cámara de vídeo y una televisión vendrían muy bien para observar repetidamente un determinado movimiento.

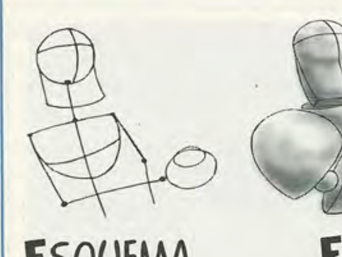
Dibujar los personajes no es más que el primer paso de una animación

¿Qué ocurre con las proporciones de la figura? A lo largo de la historia el ideal de belleza ha ido cambiando. Aparte de otros factores, en este ideal interviene la medida de la cabeza. Actualmente el canon de la figura humana tiene una altura de ocho cabezas. Esto quiere decir que la altura humana ideal resultaría de medir la altura de la cabeza y multiplicar esta medida por ocho. Además de este parámetro, en los hombres se suele estilizar la figura más si se incrementa la masa muscular y la figura femenina alargando la dimensión de las piernas... En la creación de personajes para videojuegos, la cabeza suele ser mucho más grande que la dimensión establecida por el canon, dándose personajes con 4, 3 o incluso 2 cabezas de altura. Esto se hace principalmente por dos motivos. En algunos géneros (como plataformas, estrategia, aventuras gráficas...) Los personajes suelen ser pequeños, y no pueden ocupar toda la pantalla. Para poder identificarse fácilmente, la cabeza tiene que ser más grande. Además, gran parte de la personalidad de un personaje está en el rostro. Incluso en videojuegos cuyos personajes apenas ocupan 10 pixeles (como en *Worms*), el rostro juega un papel fundamental en la expresividad de los mismos.

En animación 3D, este aspecto pierde interés, ya que el limitado número de polígonos que disponen los grafistas para sus modelos no les permite concentrarse demasiado en la cabeza pero, con la avalancha de aceleradoras 3D, esto cambiará pronto. De cualquier forma, siempre se pueden animar las texturas de la cara, consiguiendo mayor realismo y expresividad.

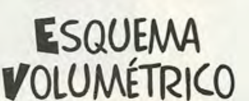
Es imprescindible crear bocetos antes de dedicarse a los diseños finales

Antes de animar el personaje definitivamente y para ahorrarnos pasos en falso, deberemos crear esquemas anteriores al diseño final que nos dejen ver si el planteamiento que tenemos pensado es el adecuado. Normalmente se acostumbra a realizar 2 esquemas; uno primario y otro volumétrico. En el esquema primario las articulaciones se representan por puntos y los huesos por alambres. Esto nos da



ESQUEMA PRIMARIO





ESQUEMA VOLUMÉTRICO



ACABADO ANATÓMICO



COLOREADO FINAL

CONSTRUCCION ANATOMICA DE NUESTRO PARTICULAR PERSONAJE, MK.





ESQUEMAS PRIMARIOS Y ANATOMICOS DE MK DISPUESTO A ATACAR AL ENEMIGO.

una idea de cómo quedará la postura que queremos dibujar y si la sensación obtenida es la buscada. Habitualmente nos acostumbraremos a hacer varios de estos esquemas (no conocemos a nadie que elija el mejor a la primera), y después de ver varias posibilidades escogeremos la que mejor nos haya quedado. Una vez elegido el que más nos gusta, pasaremos a darle volumen. En las ilustraciones que acompañan este artículo, el esquema volumétrico se ha acabado con excesivo detalle. Normalmente no nos preocuparemos de las sombras de las figuras trigonométricas, ni siquiera haremos las esferas de las articulaciones... sin embargo sí "modelaremos" en parte la figura, bien con conos y esferas o bien simplemente con círculos y óvalos, para darle volumen.
Este esquema (el volumétrico) además nos servirá para hacer una "pre-animación" rápida en nuestra mesa de luz. Captaremos de esta forma los puntos críticos del movimiento para pasar después al dibujo completo (anatómico) del personaje. Hay grafistas que prefieren dibujar al personaje sin ropa y después pintársela encima. De esta forma, tienen una base anatómica sólida sobre la que guiarse. Nosotros no "perderemos" el tiempo en esto y directamente dibujaremos el personaje completo.
A partir de este punto, para todos (o casi todos) los ejemplos del curso nos serviremos de un hipotético personaje de videojuego, un bárbaro guerrero que de aquí en adelante llamaremos "MK".

Hay que conocer al dedillo la anatomía de nuestro personaje para dibujarlo siempre igual

Tenéis varios ejemplos de esquemas primarios y volumétricos de MK en varias situaciones. Es muy importante coger soltura al dibujar distintos personajes, expresiones faciales (si el juego llevará primeros planos del personaje) y sobre todo en esquemas primarios y volumétricos. Serán la base para todo trabajo de animación tradicional. Un buen ejercicio para adquirir manejo sería realizar esquemas tanto primarios como volumétricos (uno encima de otro) de personas o dibujos que aparezcan en revistas, cómics... el dibujar con soltura será fundamental en animadores 2D ya que, para hacer el juego completo de animaciones de un personaje (sobre todo si es el que controla el jugador), se necesitan muchos *frames*. Para empezar bien con nuestro nuevo "ayudante" del curso, lo animaremos andando en nuestra primera animación con Autodesk Animator Pro.

ANIMANDO VOY... ANIMANDO VENGO...

Antes de dibujar a la perfección a MK, se hicieron unas pruebas con esquemas volumétricos sencillos en la mesa de luz.

Tras dibujar un *frame* a lápiz, no podemos descuidarnos: hay que perfilarlo a tinta

Rectificamos partes que no quedaban bien, señalamos algunos detalles que no debíamos olvidar en la animación definitiva y comenzamos a dibujarlo con detalle. ¿Con qué nivel de detalle? ¿Qué dimensiones debe tener nuestro personaje cuando lo dibujamos en papel? En las animaciones de ejemplo (que podéis encontrar en el CD de la revista), MK «mide» 98 pixeles. Originalmente en papel, los dibujos miden 7 cm de alto. Esto puede dar una medida aproximada (1 cm sería equivalente a 14 pixeles si se escanea a 300 dpi). Sin embargo, nuestro personaje tiene bastante nivel de detalle, y ello influye en el tamaño de los dibujos originales en papel. De esta forma, la relación anterior es orientativa. Cuando dibujéis un *frame* a lápiz y lo tengáis acabado, deberéis limpiar las líneas repasándolo en ese momento a tinta, sin dejar líneas que ensucien el dibujo. Todos los trazos deberán quedar claros y firmes. Esto nos



MK UN DOMINGO POR LA TARDE VIENDO LA TELE.



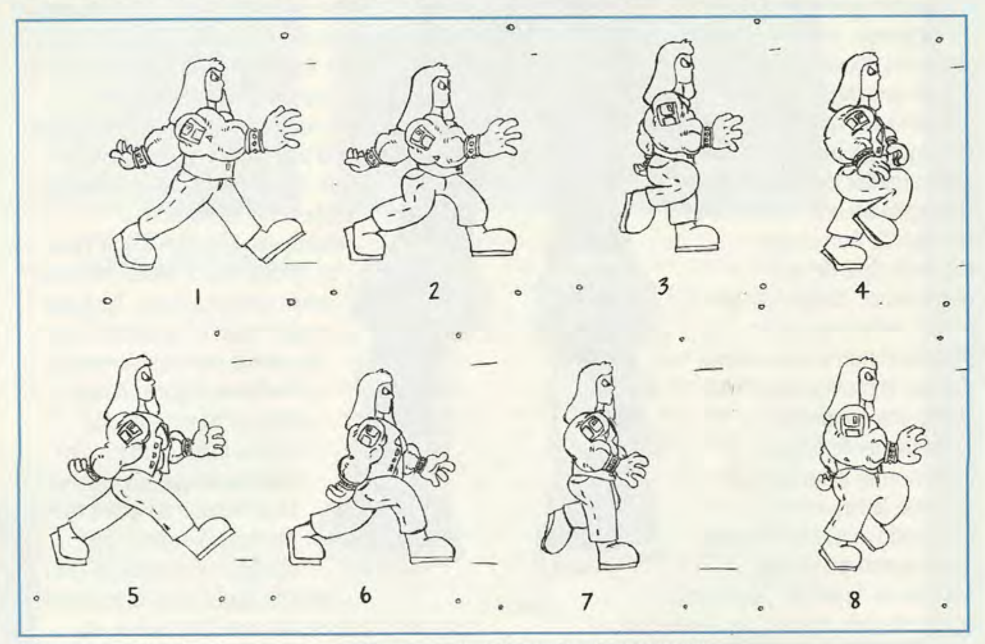
Diferentes expresiones faciales del personaje.

servirá para que, al dibujar el siguiente cuadro de la animación, las líneas del dibujo anterior no nos confundan. Aunque al principio pueda resultar costoso el tener que pasar a tinta cada *frame* y borrar las líneas de lápiz antes de pasar al siguiente, esto nos ahorrará confusiones y pasos en falso.
Con el fin de montar más fácilmente la animación en el Animator, desde el primer cuadro de la animación, dibujaremos en todos algunos puntos de referencia. En la animación de MK andando podemos observar que se han hecho 3 marcas (3 puntos) en las esquinas inferiores y en la superior derecha. Con esto, cuando trabajemos con los *frames* individuales, podremos centrarlos teniendo como referencia esas marcas. Definiendo el color de fondo como transparente (en nuestro caso el blanco), moveremos los *frames* consecuentemente fijándonos en el anterior.

Para que sea sencillo realizar el montaje de la animación, hay que situar puntos de referencia

Cuando trabajemos en la mesa de luz, un truco para tener una idea de cómo va el trabajo es levantar de una esquina el folio del *frame* en el que estamos trabajando, teniendo el *frame* anterior debajo. Realizando el movimiento de levantar y bajar por una esquina el folio superior rápidamente, podremos ver los puntos en los que falla la animación y estaremos a tiempo de corregirla antes de ponernos en el ordenador.
Una vez terminados los *frames* en papel, los escanearemos con una buena resolución. El trabajar a un tamaño mucho más grande del definitivo nos deja bastantes ventajas, como por ejemplo que al colorear se notarán mucho menos las imperfecciones (no nos preocuparemos si se nos quedan algunos espacios pequeños sin colorear), las líneas al reducirse quedarán perfectas, etc. Para el ejemplo de MK andando, se escaneó en 256 tonos de grises a 300 puntos por pulgada.

Una vez escaneadas las imágenes individuales, cargaremos Adobe Photoshop y ajustaremos el brillo y el contraste de cada una como explicamos el mes pasado. Para terminar de limpiar esos pixeles que quedan de color gris, usaremos la herramienta de tono, con la persiana *dodge* activada, y trabajando con tonos altos *highlights* pasaremos un pincel grueso por toda la imagen. Veremos cómo se eliminan todas las impurezas no deseadas,



Nuestra animacion en una unica imagen lista para ser coloreada y reducida.

Algunas aclaraciones

El programa que vamos a utilizar para nuestros ejemplos será Autodesk Animator Pro 1.0. Puede haber confusiones con una versión anterior del programa llamada Autodesk Animator 1.01, que sólo soportaba animaciones en formato FLI y tarjeta gráfica VGA. El AA Pro soporta distintos modos de vídeo (incluido 800x600, 640x480...) además de opciones para hacer nuestros propios scripts y nuevas funciones respecto de la versión anterior. Será ésta la versión que utilizaremos.
Las animaciones FLC, si se cargan con el visor SEA (incluido en el CD de la revista), se reproducirán correctamente menos el último *frame*, que se lo salta. Si vais a cargar los ejemplos de esta entrega con ese visor, las animaciones no se mostrarán suaves.
Posiblemente en sucesivas entregas del curso podamos contar con un programa de animación realizado por Lorenzo Portillo en español y en versión exclusiva para lectores de Game Over. Se están dando retoques, corrigiendo bugs y demás problemas... para finales del curso de animación posiblemente podamos incluirlo en el CD de la revista.
El mes que viene explicaremos las funciones de Autodesk Animator Pro, para que en las 5 entregas siguientes podamos meternos de lleno en la animación de personajes (no haremos más hincapié en explicar Photoshop ni Animator). Nada más... hasta el mes que viene.



quedando las líneas limpias para que podamos utilizar la herramienta de relleno y así ahorrar mucho tiempo al colorear el dibujo. Pero antes de ponernos a colorear convendría montar la animación tal como está en Animator. Aunque pueda parecer una pérdida de tiempo, así nos aseguraremos de que los *frames* están bien construidos y podremos, en caso necesario, insertarle alguno más.

Un pequeño inciso sobre el número de *frames* que debe tener la animación. En películas de animación, el número de *frames* debe ser muy alto para que sea posible conseguir movimientos fluidos. En animación para videojuegos, sin embargo, normalmente no podemos contar con todos los *frames* que queramos. La máquina a la que irá destinado el juego y las habilidades del programador de nuestro grupo nos limitarán enormemente el número de *frames* por segundo (fps) de que disponemos. Debemos, en efecto, minimizar en lo posible el número de *frames* sin que el movimiento final tenga una apariencia excesivamente brusca.

Si al montar la animación (sin que la hayamos coloreado aún) observamos que nuestro personaje todavía va "a saltos", podremos insertarle más *frames* (siguiendo todo el proceso completo: dibujarlos en la mesa de luz, escanearlos, etc.).

La animación para videojuegos no proporciona a los dibujantes libertad en el número de *frames*

Sin embargo, si nos pasamos inicialmente en el número de *frames* que hemos incluido en la animación (por ejemplo, podríamos haber hecho que MK anduviera rondando los 20 *frames*) tendremos que eliminarlos luego y habremos perdido un tiempo que en el desarrollo de un juego puede ser precioso. Un truco que es bastante utilizado en el caso de que la animación tenga muy pocos *frames* es disminuir en lo posible la velocidad de la misma (es decir, el número de *frames* que haya por segundo).

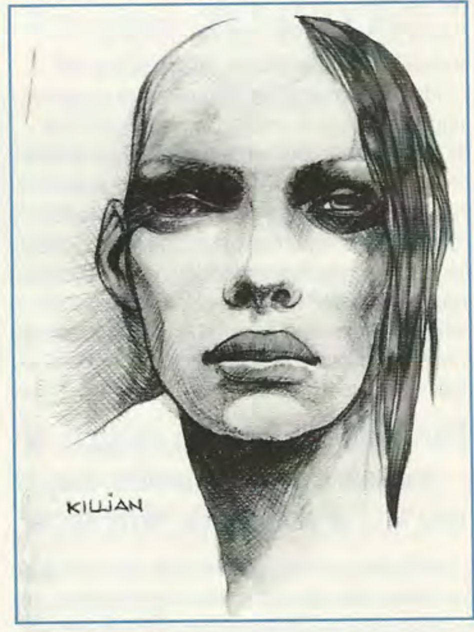
Hagamos una prueba; en la animación MKANDA.FLC vamos a aumentar el número de fps a 15 y el resultado será más brusco. Otro truco para dar la sensación de que hay más *frames* en la animación es usar líneas cinéticas y repeticiones de la misma parte del cuerpo en un frame. Por ejemplo, cuando MK anda, podríamos haber dibujado en cada *frame* 4 brazos, 2 en primer plano y 2 en segundo. Los dos brazos nuevos se adelantarían a los antiguos y servirían de "puente" entre el *frame* actual y el siguiente (veremos algunos ejemplos de esta técnica a lo largo del curso). Esto puede parecer extraño cuando se dibuja, pero al verlo todo junto da la sensación de animación más suave (y tenemos el mismo número de *frames* que antes). El videojuego *Earth Worm Jim* utiliza esta técnica en gran número de animaciones.

Una vez hecha la prueba preliminar de la animación (sin colorear) y visto que el resultado obtenido es satisfactorio, pasaremos a colorearla. El coloreado se hará en Photoshop, y básicamente se darán colores planos. Para que podamos usar la herramienta de relleno más rápidamente, deberíamos haber cerrado todas las superficies que vayan a ir del mismo color cuando dibujamos en papel. Una vez que tenemos todas las líneas limpias y los dibujos coloreados (el mes pasado explicamos la forma de colorear rápidamente con Photoshop), los juntaremos en una misma imagen. Para nuestro ejemplo, reducimos esta imagen a 640x480 pixeles y la cargamos en Animator. Allí juntamos los *frames* y montamos los ejemplos que tenemos.



LAS MARCAS SERAN UTILES A LA HORA DE MONTAR LA ANIMACION QUE TENEMOS ENTRE MANOS.





El coloreado de las animaciones se realizará con PhotoShop y con colores planos

Las funciones básicas de Autodesk Animator las explicaremos el mes que viene en esta misma sección. Podríamos haber explicado el uso del programa en esta lección, sin embargo hemos creído conveniente comenzar con el primer ejemplo en esta entrega del curso para que, cuanto antes, podáis empezar a hacer vuestras primeras animaciones. Ya sabéis; la mejor forma de aprender a caminar es ponerse a caminar.